BULGARIAN ACADEMY OF SCIENCES

CYBERNETICS AND INFORMATION TECHNOLOGIES • Volume **11**, No 2

Sofia • 2011

Development of a Refactoring Learning Environment

Mincho Sandalski, Asya Stoyanova-Doycheva,
Ivan Popchev, Stanimir Stoyanov

Plovdiv University "Paisii Hilendarski", 4000 Plovdiv
E-mails: sandim@uni-plovdiv.bg, astoyanova@uni-plovdiv.bg, sandim@uni-plovdiv.bg, stani@uni-plovdiv.bg

Abstract: *The paper describes a Refactoring Learning Environment, which is intended to analyse and assess programming code, based on refactoring rules. The Refactoring Learning Environment architecture includes an intelligent assistant, Refactoring Agent, which is responsible for the analysis and assessment of the code, written by students in real time by using a set of refactoring methods. According to the situation and based on the refactoring method, which should be applied, the agent could react in different ways. Its goal is to show the students, as much as possible, the weak points of their programming code and the possible ways of improving it.*

Keywords: *refactoring, e-learning, software engineering, agent-oriented architectures.*

1. Introduction

The existence of a large number of legacy systems and the necessity of their improvement for the purposes and needs of their users give rise to the creation of a specific process in software engineering, called reengineering. Reengineering is demanded in different situations when the software needs to undergo an evolution [1]. Here are some examples for the necessity of reengineering: the division of monolithic software into separate modules with the objective of their easier management, enhancement of the software productivity or portability, the use of new technologies or a change in the clients' requirements. When the software constantly adapts or modifies, its source code becomes more complicated and its

initial architecture and structure lose their identity. For this reason the major share of the software development price is focused on their support [2]. In the application of well-known and effective software development approaches such as iterative, evolutionary and others, a solution has not been found in connection with the code complexity. This is so because in these approaches the software engineers' efforts concentrate on developing new requirements while at the same time the software has to be supported [3]. In order to solve the code complexity problem, within the reengineering there emerged the special techniques restructuring [4] and refactoring [5].

At the Faculty of Mathematics and Informatics of the Plovdiv University, a Distributed e-Learning Center (DeLC) has been created [6, 7]. It has been considered a good practice to include special techniques for code simplification in the students' education, especially in the Master software engineering programs. In this publication we present in detail an e-Learning environment for assisting the acquisition of the special technique of refactoring, called Refactoring e-Learning Environment (ReLE). A brief overview of ReLE can be found in [21]. The rest of the paper is organized as follows: Section 2 briefly presents the refactoring technique and reviews existing supporting tools; Section 3 describes the ReLE architecture; Section 4 presents an intelligent agent, called RA, which is the kernel of the environment; in Section 5 some implementation issues have been discussed; Section 6 demonstrates the use of the RA, and finally Section 7 concludes the paper.

2. Refactoring

The main goal of refactoring is to improve the design of the existing programming code. In [5] the refactoring is presented as a sequence of small transformations of the source code in a software system, as they preserve the external behaviour of the software system and lead to large restructuring of the source code. Each transformation can be presented as a pattern that is called "refactoring". Fig. 1 presents an example of refactoring.

Extract class

Condition: You have one class doing work that should be done by two classes.

Motivation: A class has many methods and quite a lot of data. A class is too big to be understood easily.

Action: Create a new class and move the relevant fields and methods from the old class into the new one.

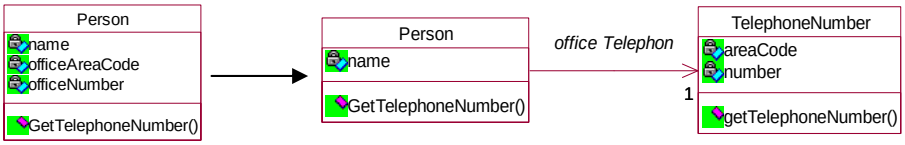

Fig. 1. Extract Class refactoring method

The activities connected with the source code transformation are defined as follows [5]:

- Localization of the "bad" places in the source code, which has to be refactored.
- Selection of the right refactoring from a refactoring pattern list according to the concrete situation in the source code.
- Provision that the applied refactoring will preserve the behaviour of the software system – we create a unit test about this place in the source code.
- Application of refactoring.
- Execution of unit tests after refactoring.
- Determining the influence of refactoring on the software quality characteristics (complexity, intelligibility, support possibilities) or on the process (productivity, price, efforts).
- Coordination support between the code to which refactoring is applied, and other software artefacts (project documents, requirements specification, tests, etc.)

Although the refactoring process could be realized by hand, the possibility of applying automatic tools is of great importance. At present, a number of such tools are available, where the aspect and degree of the process automation vary depending on the particular tool and supported maintenance. Tools like Refactoring Browser [8], XRefactory [9] and jFactor [10] apply semi-automatic approach, after which the place and type of refactoring are chosen by the user. According to some scientific researches completely automated refactoring is also an acceptable approach. Guru, for example, belongs to this category and is used for restructuring hierarchies of successors and methods for refactoring in SELF programs [11]. Some other approaches for automated refactoring are presented in [12-15]. A current tendency in this field consists in the integration of refactoring tools in powerful industrial environments for software development. Such is the case with Smalltalk Visual Works from v7, Eclipse from v2, Together Control Center from v6, IntelliJ IDEA from v3, Borland JBuilder from v7, etc. All these tools focus on applying refactoring in compliance with the user requirements. Another group of tools, which are less in comparison with the previous ones, provide the opportunity to define when and where to apply refactoring. In [16], an approach is presented after which the implementation is realized via metrics, whereas in [17] the possibility is described for automation via invariants by means of the Daikon tool. The latter approach is based on a dynamic analysis of the behaviour of the run-time of the system and its most proper application is as a complement to the other approaches.

The proposed environment differs from the existing tools in several aspects:

- The environment is a prototype and is intended, first of all, for teaching students.
- The code analysis is done in real time, i.e. during the code development phase the students can be assisted in improving its quality.
- An agent-oriented implementation is realized.

3. ReLE Architecture

The ReLE architecture consists of two components (Fig. 2):

- Front-End (FE) – the environment, which is used by the students for the development, compilation and testing of the source code;
- Back-End (BE) – the Refactoring Agent (RA), which is an intelligent agent assisting the students during the code development.

The Refactoring Agent is an autonomous software application that continuously analyses and assesses the code that is developed in FE. Consequently, from the RA point of view, FE is its environment. The Refactoring Agent communicates with its environment by means of its sensors and effectors. Via the sensors the RA accesses the complete source code. This implies not only the files being edited, but also the completed ones that have not been opened in the FE for editing. In this way the agent could make a profound analysis and give an adequate assessment for the required changes on the basis of all the code rather than the part that is currently being modified. The sensors also provide some basic metric information to the agent, which is used for initial filtering of the possible refactoring methods that can be further evaluated. The possible metrics are LOC (Line Of Code) per class/method, number of methods/attributes per class, and so on. The role of the effectors is to trigger different events that assist the students during the accomplishment of their tasks in FE, where they are working. Such events could be:

- Underlying particular parts of the code by highlighting them with an appropriate colour;
- Displaying messages in dialog windows, balloon messages, etc.;
- Emitting sound-signals, vocal messages;
- "Incarnating" the agent in the form of animation to exalt the effect.

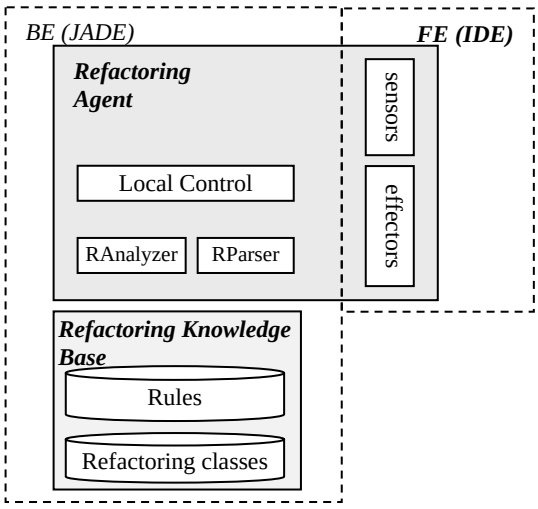

Fig. 2. ReLE architecture

The collaboration of the sensors and effectors is coordinated by the Local Control of the agent, which is based both on the information, incoming from the sensors, and the refactoring rules, stored in the Refactoring Knowledge Base (RKB) of the agent.

The analysis of the source code, written by the students in FE, is made by the RAnalyzer. Before the RAnalyzer starts its work, the RParser parses the source code and creates a tree structure from it. This tree structure can be analysed by the RAnalyzer.

The RKB consists of a set of rules together with a set of classes, which build a consistent knowledge base. Each rule describes in a common form the conditions, which allow a particular refactoring method to enter the "short list", based upon some metrics.

For example, a possible rule for choosing the "Extract class" refactoring method could be LOC_by_class > predefined_value, which actually means that the refactoring method will be fired when the class becomes too big (depending on the predefined value).

In this way, the rules are used by the RAnalyzer in order to make the initial filtering of the refactoring methods, which should be evaluated at the next step.

Each refactoring class contains the code for the particular refactoring method, as well as a code for the final evaluation of the possibility of applying this refactoring method. The refactoring methods filtered by the RAnalyzer are then examined by using the evaluation part of each refactoring class. In this way, the agent takes a final decision about which refactoring method to be used at what place.

At the last step, the refactoring is applied by using the actual refactoring class after a negotiation with the user – as described in the next topic.

4. Refactoring Agent RA

As mentioned, the kernel of ReLE is an intelligent agent assisting the students in the process of refactoring. Its main task consists in checking the code, which is being developed by the students in FE, and appropriately displaying instructions for improving its quality, whenever necessary. Depending on the refactoring method, which should be applied, the agent could react in three different ways (Fig. 3):

- **Automatic Refactoring** – to apply the method automatically after receiving a confirmation from the user.
- **Refactoring Proposal** – to display detailed instructions, explaining to the user where and how the particular refactoring method should be applied.
- **Refactoring Questionnaire** – to ask the user additional questions in order to clarify the conditions and define the appropriate refactoring method.

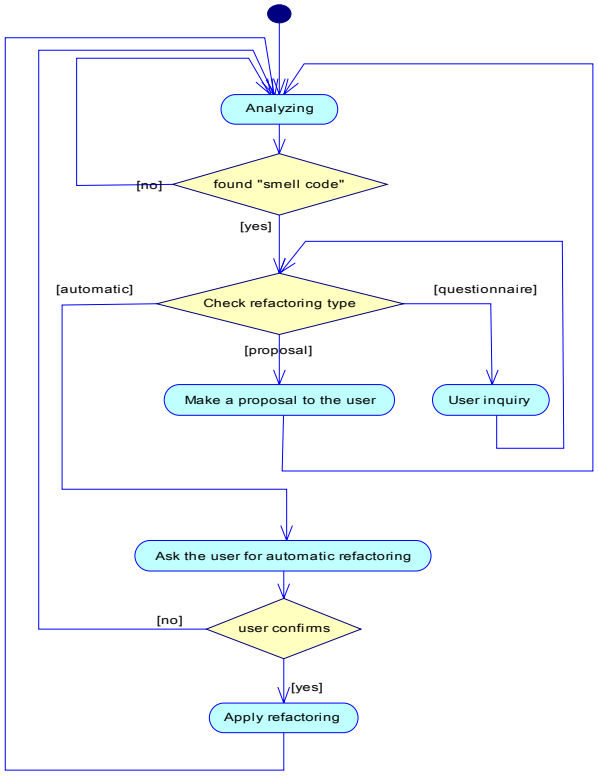

Fig. 3. Activity diagram of RA functionality

In the cases when the refactoring method is comparatively simple and the criteria for its application are clear, the agent could offer the user to implement the required changes automatically. Some of the appropriate methods for this approach are: Move Method, Move Field, Extract Class, Extract Method, etc. Here is an example (Move Method):

- On the basis of the refactoring rules in RRB, the agent finds out that there is a method in class A, that uses resources mainly from class B, which implies the application of the Move Method.
- The agent displays a message, in which it offers the user to move this method to class B.
- In case the user agrees, the agent moves the whole method from class A to class B by:
  - correcting all references to the resources in class A, so that they are accessible from class B, and adding the required parameters to the method if needed;
  - substituting all method invocations, so that they use its new position in class B.

The intervention of the user in this process is not excluded, of course, and it is possible in case he/she would like to correct the code of the method after it has been replaced in the other class.

Often, the criteria for refactoring are clear but the application of the particular method implies a significant change in the code or its structure. In these cases an approach is recommended, after which the agent informs the user about the specific situation and offers them detailed explanations about the possible improvements that could be made in the particular situation. Some of the proper refactoring methods that belong to this category are Replace Conditional with Polymorphism, Replace Delegation with Inheritance, Replace Inheritance with Delegation, etc. Here is an example (Replace Conditional with Polymorphism):

- In compliance with the rules in RRB, the agent finds out that there is a condition, which chooses a different behaviour on the basis of an attribute of a given object. This is an appropriate case for applying the method Replace Conditional with Polymorphism;
- The agent underlines this part of the code and displays a balloon message, which points out that refactoring of the marked code is required;
- If the user would like to become familiar with some additional information concerning the agent's recommendation, such could be presented in the form of consistent steps, which he/she should implement in order to improve the quality of the code.

Often the choice of applying one or another method for refactoring is made on the basis of an almost uniform set of criteria where just a few differ from one another. In the cases when some of the requirements for applying the refactoring methods are met and yet this is not sufficient to define uniquely the most appropriate one, the agent could "ask" the user several questions in order to clarify the concrete situation. Having clarified the requirements, the agent defines again the type of the situation, which could be one of the types described above: automated refactoring or a refactoring proposal. Here is an example:

- The agent finds out that a given class contains a numeric "type code" but cannot determine if this code changes the behaviour of the class;
- The agent displays a question to the user, asking them whether this "type code" influences the behaviour of the class;
- If the answer is "no", the agent offers the user to apply the Replace Type Code with Class method;
- If the answer is "yes", the agent asks the user whether the "type code" attribute changes during the lifecycle of the object;
- If the user's answer is "no", the agent offers the user to apply the Replace Type Code method with Subclass;
- Otherwise, the agent offers the Replace Type Code with State/Strategy.

5. Implementing the RA's prototype

A brief overview of the RA's prototype is described in this section. It is a big challenge to create an integrated development environment, consisting of the RA's environment and the working environment for the students. Furthermore, the

implementation of the RA's internal architecture and the embedded knowledge base are presented in the section.

5.1. Integrated development environment

The integrated development environment consists of an agent-oriented development tool – JADE [19], and an environment for developing Java programs – Eclipse [18]. Both environments have a rich set of integration features, providing a possibility to work together. On the one hand, Eclipse uses the "plugins" concept, which brings a powerful mechanism for interaction with external components. On the other hand, the agents' sensors and effectors are the environment interaction mechanism. This raised the idea of both the sensors and the effectors to be developed as plugins in Eclipse. In this way, one of the significant concepts in the agent-oriented systems is implemented with Eclipse components. As a result, the integration between the two environments was implemented entirely by their own tools.

The RA's implementation is distributed in both environments – JADE and Eclipse (Fig. 4). Some of the Java classes, implementing RA, are built in the Eclipse Platform in the form of plugins. In this way, the RA has access to the needed data (Resources) in Eclipse – mainly this is the source code, produced by the students. The agent can interact with the graphical User Interface (UI) as well as other APplication Interfaces (API). In this way the communication between the student and the RA is implemented.

The RA has to reside a particular JADE container. During the RA operation different behaviours can be added to the agent. Each behaviour implements logic for the analysis and modification of the source code. In this way each refactoring method is presented as separate behaviour. Furthermore, the RA can communicate with other agents in the same container and assign them tasks, using JADE API. The RA's access to Eclipse Java Editor is possible by the help of Java Development Tools (JDT) plugin.

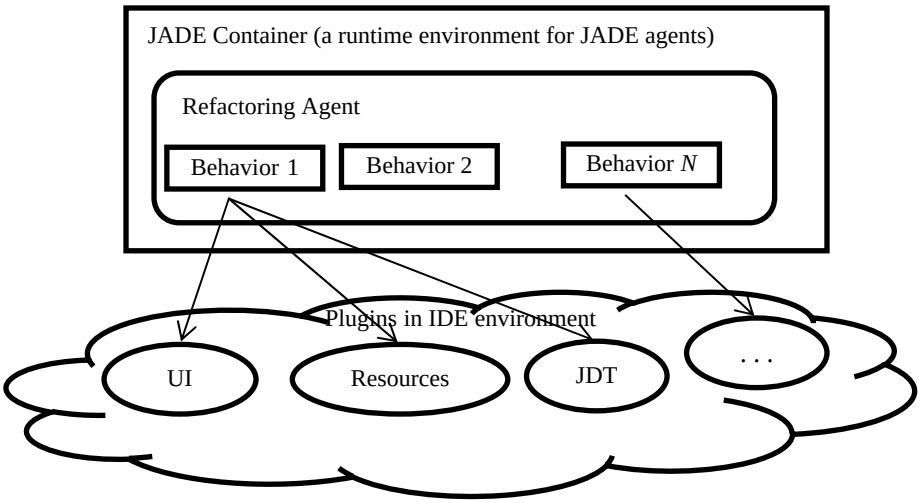

Fig. 4. Integrated development environment

5.2. RA knowledge base

The knowledge base of the Refactoring Agent is a fundamental part of its architecture. It contains a set of rules determining whether a situation requires refactoring and which refactoring methods should be applied.

Each class of the agent's knowledge base on refactoring includes a code, which implements a specific refactoring method, and a code, enabling the final evaluation of the possibilities for applying the refactoring method (the evaluation part). The refactoring methods, chosen by the RAnalyzer (Fig. 2), are under investigation. For that purpose there is used the evaluation part of the classes, which implement the refactoring methods in the knowledge base. In this way the agent takes a final decision about which refactoring method to use and in which location to apply it in the code, written by the student.

The current implementation of the RA knowledge base is presented on the package diagram in Fig. 5. The main package in the knowledge base, called "Pattern", contains the common functionality for the rules and for the refactoring methods. Each refactoring method is a set of classes that extends the abstract functionality from the "pattern" package in such a way as to reach the needed refactoring behaviour. The set of classes of the refactoring method is put in a different package for each refactoring method. Currently the knowledge base contains packages for seven different refactoring methods, represented with a separate package on the diagram in Fig. 5.

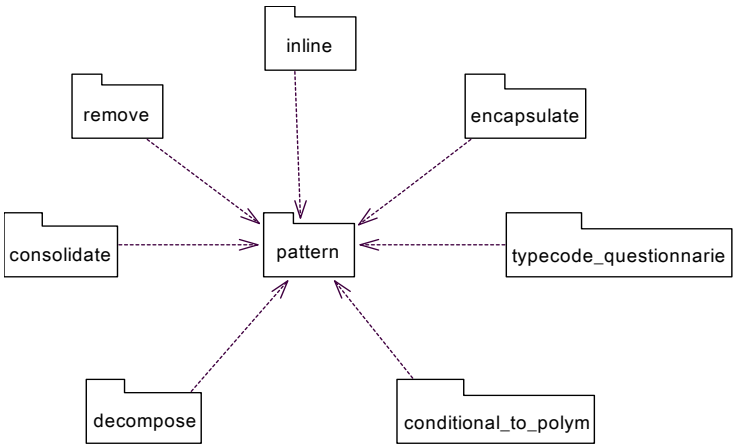

Fig. 5. Package diagram of the RA knowledge base

The structure of the refactoring packages in the knowledge base is illustrated by the "decompose" package (Fig. 6). The classes on this class diagram correspond to the classes in Fig. 5, but here they are shown in a more specific way. The Visitor class traverses the current syntax tree and looks for package specific situations in the code, and the Resolution class, which contains the logic for the agent reaction for this particular package.

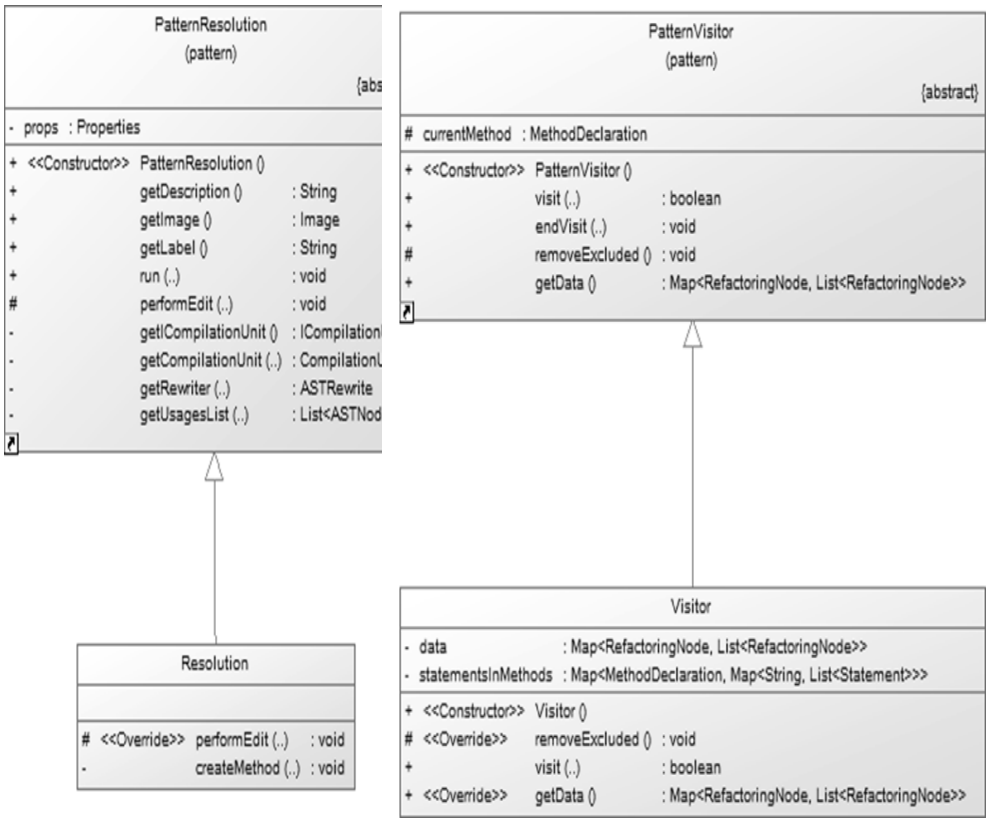

Fig. 6. Implementation of the refactoring method

The structure of the other refactoring packages is similar. The implementation of a new refactoring method needs two additional classes that implement the concrete behaviour.

5.3. RA Components

The major classes and their relationships, implementing the RA, are shown as a class diagram (Fig. 7). The RA life cycle is depicted in the sequence diagram (Fig. 8). The main steps of the life cycle are described in more details.

The first step is the initiation of the RA (1-5 in the diagram). The Activator class creates a JADE container, in which the RA operates. This class contains RALifecycleListener objects, the so called observers/listeners (in the Observer pattern [20]), and methods for notification of all registered observers. Each refactoring method corresponds with a separate observer. The observers inherit the MarkerManager class used for highlighting code fragments, considered for refactoring. The highlighting option can be deactivated by the student through the start/stop button of the RA.

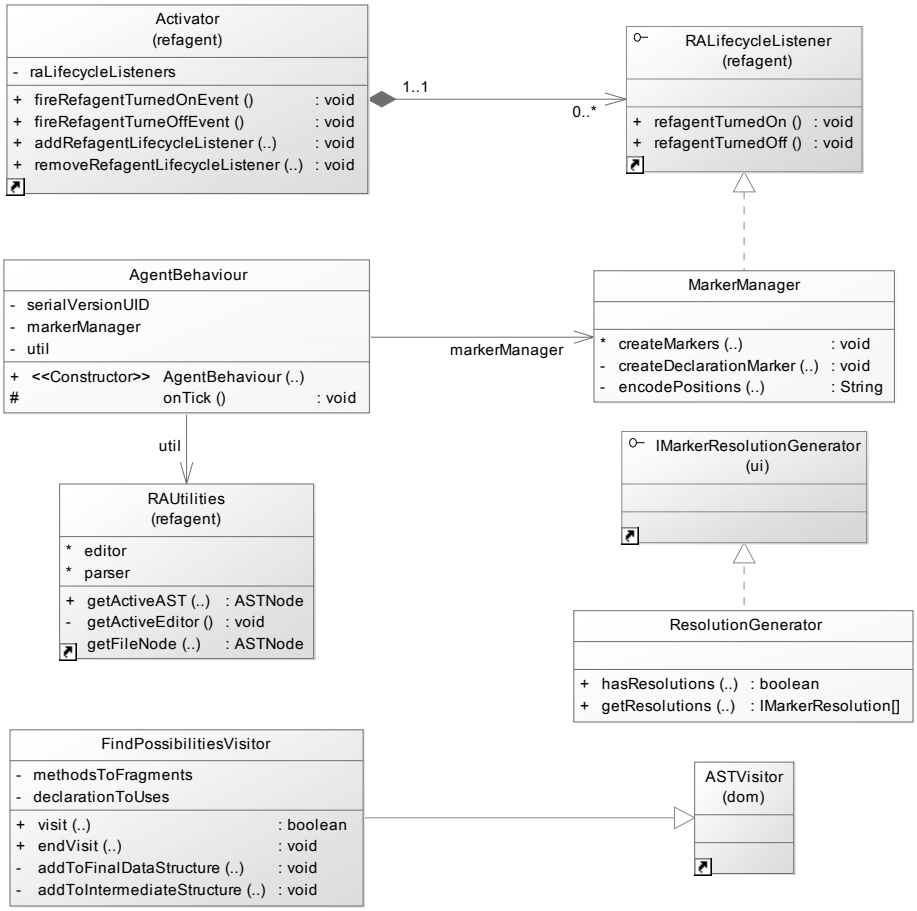

Fig. 7. RA class diagram

After the initialization of a JADE container, the control is passed to the agent. The refactoring methods are implemented as instances of the AgentBehaviour class. In the second step the current Java program is transformed into a syntax tree (AST) by RAUtilities, which uses the JDT plugin in Eclipse (mark 6 in the diagram). The syntax tree is kept continuously in conform to the actual state of the source code by the JDT plugin. Furthermore, the agent works with the syntax tree to localize the places suitable for refactoring. In order to generate the syntax tree there is used the Java pattern shown as a class diagram in Fig. 9. In the third step a Visitor is created, which crawls the syntax tree and fills in the data structures, unique for each refactoring situation (marks 8-11). The agent determines its reaction according to the gathered information. It can react in one of the following three ways: automatic refactoring, refactoring proposal, and refactoring questionnaire. The fourth step (mark 12) is for the creation of markers, which are visually represented as underlined parts of the source code. This is possible because each AST node contains actual and detailed information about the position of the corresponding syntax element. In this way the student attention is focused on the most interesting parts of the source code.

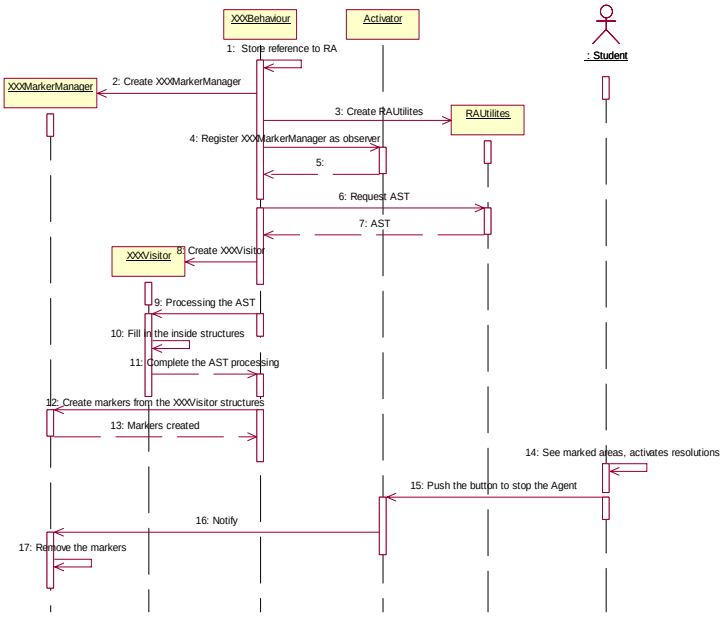

Fig. 8. RA sequence diagram

Until now we have not discussed the way in which the RA's behaviour reacts with the automatic refactoring, the refactoring proposal or refactoring questionnaire. This is because the connection between the JADE-behaviour and the so called marker resolutions is not direct. It is set by a configuration file. In principle a class is created, in which the logic's reaction is encapsulated. This logic is executed by the user when they points with the mouse cursor at the marker's annotations and extract the data, needed for its execution, from the marker's attributes.

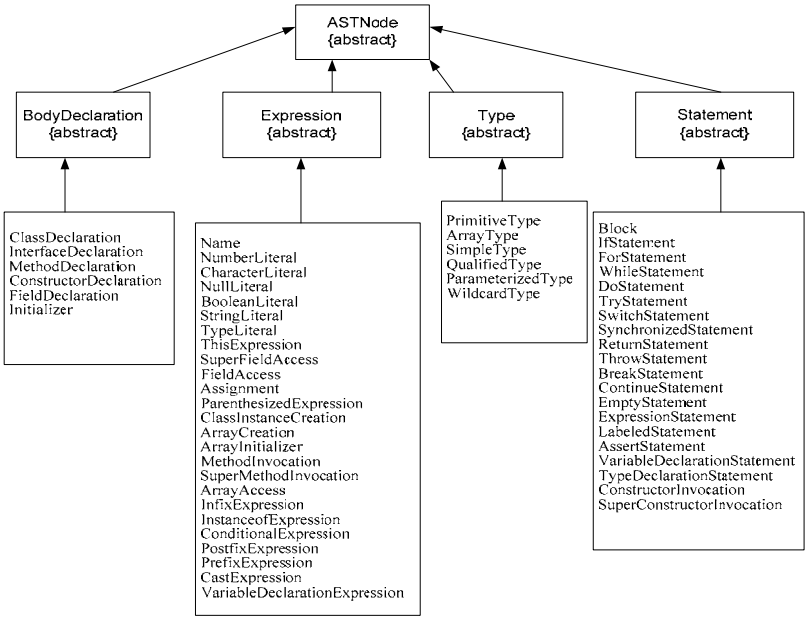

Fig. 9. AST class diagram

6. RA based student teaching

The RA agent can be used for teaching students in the following three scenarios:

- Inline Temp
- Replace Conditional with Polymorphism
- Typecode Questionnaire Behaviour

6.1. “Inline temp” scenario

An example for automatic refactoring behaviour of the RA is the “Inline Temp” situation. By definition, this situation occurs when a temporary variable is used to hold the result of a simple expression, mostly the result of a method call. According to Fowler’s guides [5] this refactoring method can be described as: *“Replace all references to that temp with the expression”*.

In cases when the refactoring method is relatively simple and the criteria for its execution are unambiguous the agent could offer the student to do the required changes automatically. In this situation the student has to:

- Accept the offer from the agent, because they are not sure what to do with the code or just likes the agent’s proposal – in this situation the agent is an assistant, that helps the student with refactoring rules;
- Do not accept the agent’s offer, because the idea, which lies behind the code, would be ruined – in this situation the student evinces creativity and the agent only shows them a piece of advice, according to the refactoring rules in the Refactoring Rules Base.

From the implementation point of view, in the scenario “Inline Temp“ the RA looks for local variables, which can be removed from the program code. The residence of the RA agent in the integrated environment is indicated by a toggle button appearing on the toolbar (in the frame) (Fig. 10). In this time the Eclipse environment is launched.

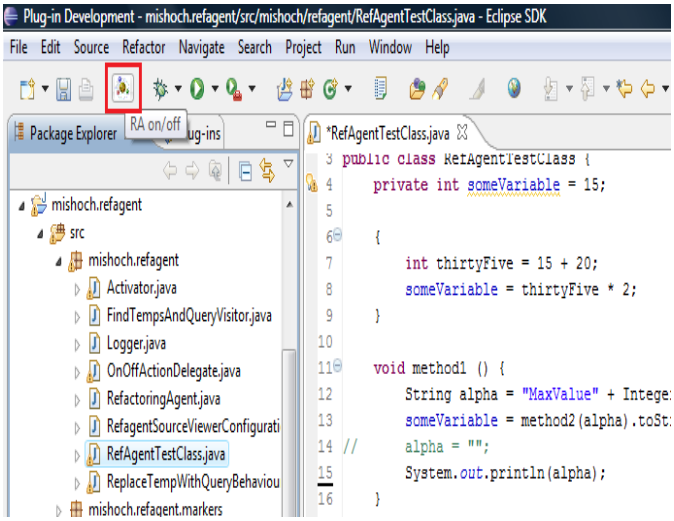

Fig. 10. The refactoring agent’s toggle button

When clicked for the first time after launching Eclipse, the toggle button creates and initializes a JADE container, which the Refactoring Agent resides. A repetitive behaviour is then added to the agent, where every 5 seconds the agent's environment (source code in the active Java editor) is scanned, and a syntax tree is generated. The syntax tree is searched for local variables that could be inline objects of a declaration. These are in fact nodes, modelling the underlined syntax construction. The variables, which are reassigned a value after the initialization, are not considered. When local variables, which the "Inline Local Variables" refactoring method can be applied to, are discovered, they are highlighted in the editor by changing their background (Fig. 11). In this way the student, working with the Java source code, can see them.

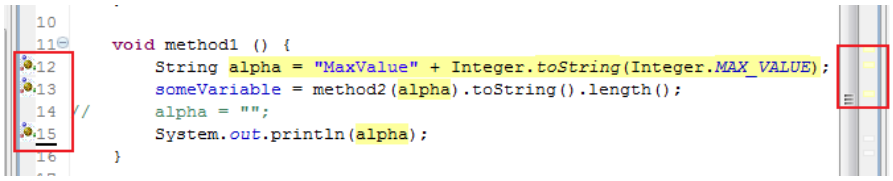

Fig. 11. Highlighted code in the editor

Furthermore, on the left vertical ruler, the Refactoring Agent's icons appear for every line, containing either the declaration or a usage of a local variable suitable for in-lining. On the right vertical ruler there appear markers (see Fig. 10) that, when clicked, scroll the editor to the corresponding line of code. When any of the icons on the left vertical ruler are clicked, the corresponding code is selected and there appears a dialog with options (Fig. 12). The first option is the one offered by the Refactoring agent.

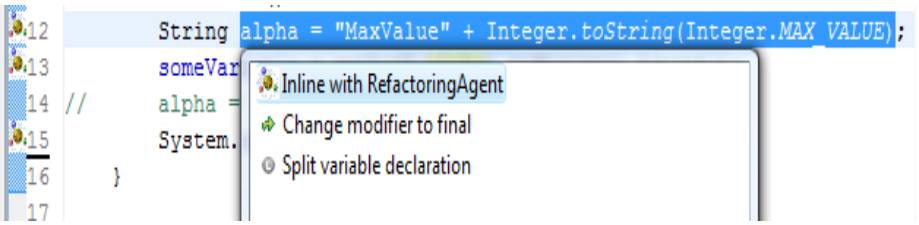

Fig. 12. A dialog window

When double-clicked, this option inlines the local variable - the declaration is removed and its usages are replaced with its value (Fig. 13).

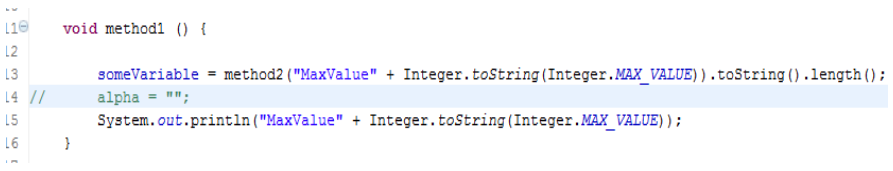

Fig. 13. Replacing the declaration with its value

The information, used to perform this action, for example a location in the source code, is obtained from the generated syntax tree. If the toggle button on the Eclipse's toolbar is pressed again, the agent's behaviour is suspended until the

button is pressed once again. The highlighting of the code stops and the icons and markers on the left and right vertical rulers disappear.

6.2. “Replace conditional with polymorphism” scenario

Often the refactoring criteria are clear but the execution of the particular method implies a significant change in the code or its structure. An example for such situation is the “Replace Conditional with Polymorphism” situation where there is a “conditional that chooses different behaviour depending on the type of an object” [5]. For this situation the refactoring method looks quite complex: *“Move each leg of the conditional to an overriding method in a subclass. Make the original method abstract.”*

In such cases the recommended approach is for the agent to inform the student about the specific situation and to propose to them detailed explanations about possible improvements that could be made in the particular situation. The student’s possible decisions in this situation are:

- To execute the proposed refactoring method – in this situation the agent guides the student by showing them a detailed list with steps for that particular refactoring method.
- To refuse the proposed refactoring method – the idea of the current code of the student is different and the suggestion of the refactoring agent is not appropriate for it. In this situation the refactoring agent only assists the student by showing them a possible decision.
- To use the proposed refactoring method but having modified it in the appropriate manner according to the particular case in the code and the concrete goal – in this situation the student evinces creativity, because they use current information about the refactoring method, proposed by the refactoring agent, and applies additional knowledge to resolve the problem.

From the implementation point of view the resulting data structure, which is to be obtained by processing the syntax tree, is a list of the “switch statements” tree nodes. The node is the syntax construction, or the switch. Furthermore, the expression has to receive as a switch condition a variable from the integer type, which is a class field (again for simplicity sake). The student’s attention is attracted by an annotation in the Java editor of the corresponding code part. By clicking on the icon, through which the annotation is designated, the refactoring proposal view opens (Fig. 14), showing an HTML document. The reason for this, as we have already mentioned, is that the Replace Conditional with Polymorphism method is too complicated for execution so it is advisable to present the student with a detailed list of steps, which they need to follow in order to apply it in practice (the steps are also the contents of the displayed HTML document). That would be much more useful for the student’s education than an automated refactoring even without considering the complexity of its implementation. The algorithm for obtaining the resulting list is simplified as much as possible. The only case, taken into consideration, is when the conditional expression has taken the form of a switch construction.

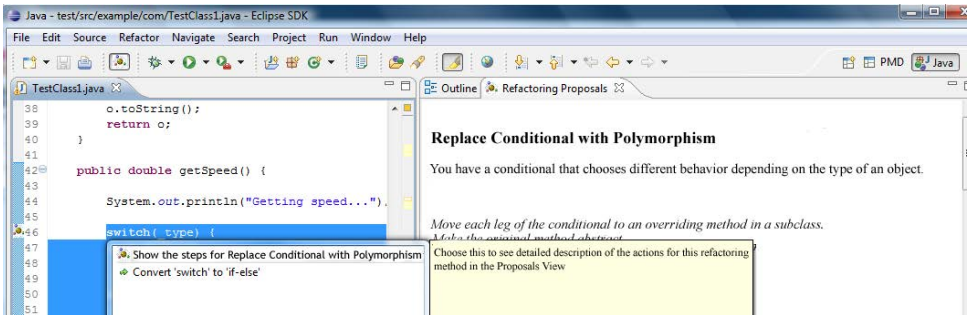

Fig. 14. An HTML file with explanations concerning the Replace Conditional with Polymorphism, visualized in the RefactoringProposalsView

The only technical peculiarity of this behaviour is that the HTML files, which have to be presented to the student and are located in a resources subfolder of the project (see the "Resources organization" section), serve as a pattern. In these files there can be placed variables, which would obtain a value when visualized. For this purpose the variables have to be surrounded by the symbols ${ and } (for example, ${TYPECODE}).

6.3. "Typecode questionnaire behavior" scenario

There are cases, where additional information is needed for the choice of applying a refactoring method, obtained by means of a dialog with the student. For example the refactoring situations "Replace typecode with class, subclasses or state/strategy" need to be "discussed" with the student in order to recognize the correct refactoring scenario. After the requirements become clear the agent defines the type of the situation again. It could then be brought to one of the types described above: automatic refactoring or refactoring proposal. When the agent is in the refactoring questionnaire situation, the student's response can be one of the following:

- To answer the question that is asked by the refactoring agent. According to the student's answer the refactoring agent offers them a particular refactoring method that belongs to the refactoring proposal, or an automatic refactoring. The response of the student depends on different possibilities, described in the previous situations;
- Not to answer the question, asked by the refactoring agent – based on the asked question, the student can make a decision that the code is clear and there is no need to be refactored. In this situation the refactoring agent helps the student only by asking the question. The student is given the chance to think about the problem based on the content of the question. The student evinces creativity and they can resolve the problem without the proposal of the refactoring agent.

From the implementation point of view for the discussed situation (Replace typecode with class, subclasses or state/strategy) the resulting structure from the processing of the syntax tree is again as simple as possible – a list of variable declaration fragments, corresponding to class member variables of the integer type, the names of which contain a substring of the "mode", "type", "class", "kind", "group", "variant", "variation", and "code" set. This criterion serves to determine

situations, where it is not completely clear which of the refactoring methods is applicable:

- Replace Typecode with Class;
- Replace Typecode with Subclasses;
- Replace Typecode with State/Strategy.

The characteristic feature of this behaviour is expressed in showing a questionnaire (Fig. 15), which is implemented as a JFace wizard as a result of the student's choice of an option from the marker context menu.

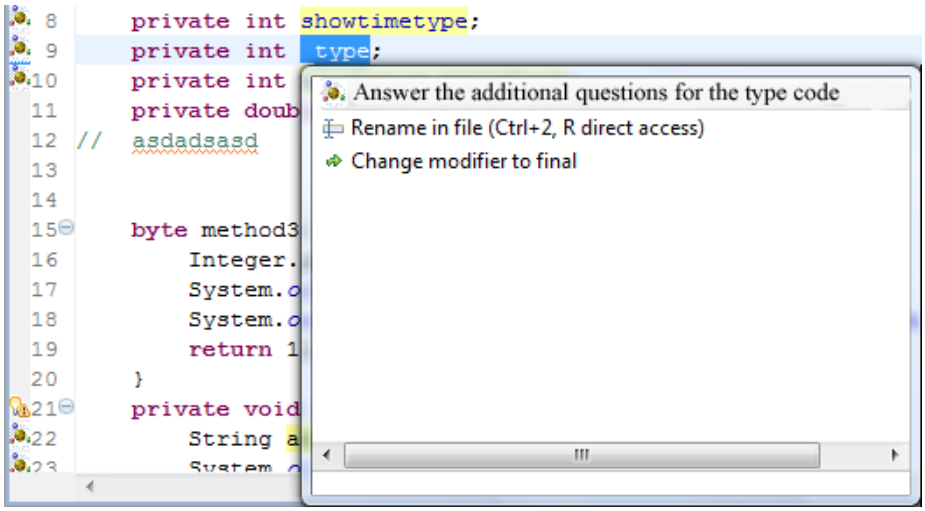

Fig. 15. A context menu for opening of a refactoring questionnaire

The questionnaire consists of two questions and three possible refactoring methods depending on the answers (Fig. 16).

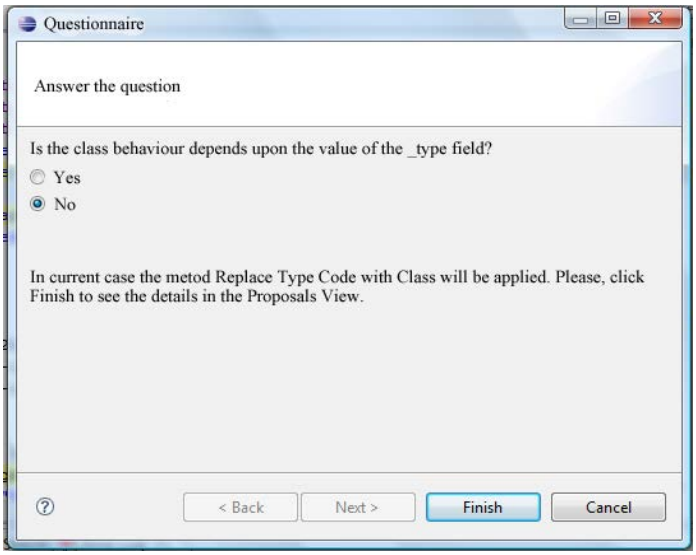

Fig. 16. The first page of the refactoring questionnaire

The answers to the questions, given by the student, are stored and serve as a basis for the selection of the exact refactoring method, which is to be displayed in the refactoring proposals view. The state (active/inactive) of the Next, Back and Finish buttons depends of the same answers. The second question from the questionnaire, if the answer to the first one has been positive, is shown in Fig. 17.

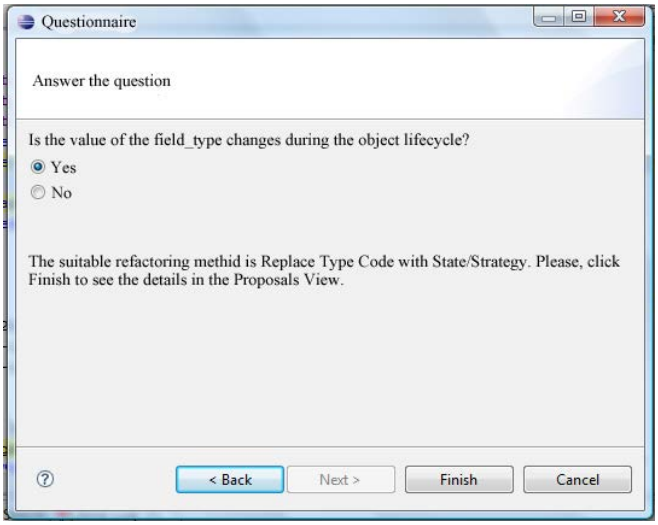

Fig. 17. The second question from the refactoring questionnaire. The answer to the first one was “Yes”

Conclusion

The implementation of the Refactoring Agent shows that the two environments – Eclipse and JADE – can work together and that their APIs could be exploited to provide the wanted behaviour:

- constant analysis of the source code on site (within the Java editor of Eclipse);
- highlighting the portions of code, which could be refactored, according to the analysis results, and proposing options to the student;
- changing the source code.

In future, the Refactoring Agent should be augmented with more logic for locating portions of source code suitable for refactoring and for providing options to resolve these situations.

The user should be able to see and navigate to all the portions of code which are considered suitable for refactoring (by means of an Eclipse “view”), as well as to ignore any of those portions, so that they are not highlighted anymore. The analysis should occur immediately at the source loading and after that in response to changing the source code by the user (i.e., writing a new code), so that it does not take up too many resources.

Acknowledgment: The authors would like to acknowledge the support of the Bulgarian Ministry of Education and Science for Research Project Ref. No ДО02-149/2008.

References

1. M e n s, T., S. D e m e y e r. Software Evolution, Berlin, Springer-Verlag, 2008.
2. C o l e m a n, D., D. A s h, B. L o w t h e r, P. O m a n. Using Metrics to Evaluate Software System Maintainability. – IEEE Computer, Vol. **27**, 1994, No 8, 44-49.
3. T a h v i l d a r i, L., K. K o n t o g i a n n i s. A Methodology for Developing Transformations Using the Maintainability Soft-Goal Graph, – In: Proc. Working Conf. Reverse Engineering, IEEE Computer Society, 2002, 77-86.
4. C h i k o f s k y, E. J., J. H. C r o s s. Reverse Engineering and Design Recovery: A Taxonomy. – IEEE Software, Vol. **7**, 1990, No 1, 13-17.
5. F o w l e r, M. Refactoring: Improving the Design of Existing Programs. Addison-Wesley, 1999.
6. S t o y a n o v, S., I. G a n c h e v, I. P o p c h e v, M. O'D r o m a. From CBT to e-Learning. – Journal Information Technologies and Control, 2005, No 4, Year III, 2-10.
7. S t o y a n o v, S., I. P o p c h e v, O. R a c h n e v a, A. R a c h n e v. DeLC – Technological Environment Supporting the Transition from CBT to e-Learning. – International Scientific Conference "Informatics in the Scientific Knowledge", 28-30 June 2006, Varna Free University, 113-127.
8. R o b e r t s, D., J. B r a n t, R. J o h n s o n. A Refactoring Tool for Smalltalk. – Theory and Practice of Object Systems, Vol. **3**, 1997, No 4, 253-263.
9. XRef-Tech, XRefactory, 2002.
   http://xref-tech.com/speller/
10. Instantiations, jFactor, 2002.
    http://www.instantiations.com/jfactor/
11. M o o r e, I. Automatic Inheritance Hierarchy Restructuring and Method Refactoring. – In: Proc. Int'l Conf. OOPSLA '96, ACM SIGPLAN Notices, 1996, 235-250.
12. C a s a i s, E. Automatic reorganization of Object-Oriented Hierarchies: A Case Study. – Object Oriented Systems, **1**, 1994, 95-115.
13. C i n n è i d e, M. Automated Application of Design Patterns: A Refactoring Approach. Ph. D. Thesis, Department of Computer Science, Trinity College, University of Dublin, 2000.
14. J a h n k e, J. H., A. Z ü n d o r f. Rewriting Poor Design Patterns by Good Design Patterns. S. Demeyer, H. Gall, Eds. – In: Proc. of ESEC/FSE '97 Workshop on Object-Oriented Reengineering, Technical University of Vienna, 1997, Technical Report TUV-1841-97-10.
15. S c h u l z, B., T. G e n s s l e r, B. M o h r, W. Z i m m e r. On the Computer Aided Introduction of Design Pattern into Object-Oriented Systems. – Technology of Object-Oriented Languages and Systems, 1998, 258-267.
16. S i m o n, F., F. S t e i n b r ü c k n e r, C. L e w e r e n t z. Metrics Based Refactoring. – In: Proc. European Conf. Software Maintenance and Reengineering, 2001, 30-38.
17. K a t a o k a, Y. M., D. E r n s t, W. G. G r i s w o l d, D. N o t k i n. Automated Support for Program Refactoring Using Invariants. – In: Proc. of the International Conference on Software Maintenance, 2001, 736-743.
18. Eclipse.
    http://www.eclipse.org
19. Java Agent Development Framework.
    http://jade.tilab.com/
20. G a m m a, E., R. H e l m, R. J o h n s o n, J. V l i s s i d e s. Design Patterns: Elements of Reusable ObjectOriented Software. Reading, Massachusetts, Addison-Wesley, 1995.
21. S t o y a n o v, S., A. S t o y a n o v a - D o y c h e v a, I. P o p c h e v, M. S a n d a l s k i, ReLE – A Refactoring Supporting Tool. Compt. Rend. Acad. Bulg. Sci., Vol. **64**, 2011 (to be printed).